# Analyzing Checkpointing Trends for Applications on the IBM Blue Gene/P System

H. Naik, R. Gupta and P. Beckman
Mathematics and Computer Science Division
Argonne National Laboratory
{hnaik, rgupta, beckman}@mcs.anl.gov

***Abstract*—Current petascale systems have tens of thousands of hardware components and complex system software stacks, which increase the probability of faults occurring during the lifetime of a process. Checkpointing has been a popular method of providing fault tolerance in high-end systems. While considerable research has been done to optimize checkpointing, in practice the method still involves a high-cost overhead for users. In this paper, we study the checkpointing overhead seen by applications running on leadership-class machines such as the IBM Blue Gene/P at Argonne National Laboratory. We study various applications and design a methodology to assist users in understanding and choosing checkpointing frequency and reducing the overhead incurred. In particular, we study three popular applications—the Grid-Based Projector-Augmented Wave application, the Carr-Parrinello Molecular Dynamics application, and a Nek5000 computational fluid dynamics application—and analyze their memory usage and possible checkpointing trends on 32,768 processors of the Blue Gene/P system.**

## I. Introduction

The past two decades have seen tremendous growth in the scale, complexity, functionality, and usage of high-end computing (HEC) machines. The Top500 [1] list shows that performance offered by high-end systems has increased by over eight times in the past five years. Current petascale machines consist of hundreds of thousands of hardware components and complex software stacks. Future exascale systems will exponentially increase this complexity, resulting in systems with hundreds of millions of hardware components. As failure rates increase with the hardware and software component count and software complexity, large-scale faults become unavoidable. This situation has led to fault tolerance re-emerging as a prominent issue for these systems.

Various methods for fault tolerance have been researched in the recent past. Fault tolerance at the hardware level has been achieved mostly through redundancy or replication of resources. Examples of this approach span from physical redundancy (RAIDed disks [2], backup servers) to information redundancy (ECC memory codes, parity memory). From the vasts amounts of research devoted to software fault tolerance, checkpointing [3] has become a popular method for achieving fault tolerance in the software. Checkpointing requires saving the local state of a process at a specific time and then rolling back (recovery) to the latest saved state in the event of a crash during the execution lifetime of a process. Since the checkpointing process impacts the overall process running time, researchers have sought ways to optimize checkpointing [4] [5] [6].

New petascale machines, such as the IBM Blue Gene series [7], offer enormous computational power. Large-scale applications using these machines continue to use checkpointing for proactive fault tolerance. Saving states of thousands of processes during checkpoint can result in a heavy demand of I/O and network resources. Since these machines have limited I/O and network resources, frequent checkpoint, at this scale, can result in longer execution times, especially if the I/O or network resources become a bottleneck. For applications to use these machines in the most efficient manner, we must understand the feasibility of checkpointing and memory trends of these applications on petascale machines. While measurement and analysis of memory trends have received some study, investigations have been limited to small-scale systems of up to 64 processes [8]. Challenges for checkpointing exhibited by large leadership machines are completely different and on a different scale.

In this paper, we study similar checkpointing trends for applications on the IBM Blue Gene/P system at Argonne National Laboratory . Specifically, we present memory trends of three popular applications: Grid-Based Projector-Augmented Wave application (GPAW), Carr-Parrinello Molecular Dynamics application (CPMD), and a Nek5000 computational fluid dynamics application. We also present an analytical model for efficiently computing the optimum checkpoint frequencies and intervals, and we determine the limitations of checkpointing such applications on large systems. Our study and analysis are based on a "full checkpointing" technique, in which the entire program state of the processes is stored during the checkpoint operation. This technique is used by the IBM Checkpointing library [9] and is the only checkpointing software currently available for the IBM Blue Gene/P machines.

Our study was conducted on 32,768 cores of the Argonne BG/P system. In the future, we plan to run the applications with full checkpointing as well as "incremental checkpointing" on all 163,840 nodes of the BG/P system.

The rest of the paper is organized as follows. In Section II,

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This research also used resources of the Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

we summarize the related work done on checkpointing in HEC systems. In Section III, we briefly describe the IBM Blue Gene system. In Section IV, we discuss the applications being used in our paper. In Section V, we present the experiments performed and analyze the results. In Section VI, we present our conclusions and outline future work.

## II. Overview of Checkpointing

Checkpointing methods and optimization techniques have been studied and summarized by several researchers [10] [11] [12]. In this section, we briefly discuss some checkpointing concepts and techniques common to distributed computing environments. In distributed systems, checkpointing can typically occur at the operating system level or the application level.

*Operating system-level checkpointing:* Operating system (OS)-level checkpointing is a user-transparent way of implementing checkpointing. In this approach, the user typically needs to specify only the checkpointing interval, with no additional programming effort; other details such as checkpoint contents are handled by the operating system [13]. OS-level checkpointing for an application involves saving the entire state of the application, inclusive of all processes and temporary data, at the checkpoint time. Since this type of checkpointing does not take into account the internal characteristics and semantics of the application, the total size of the checkpointing data can dramatically increase with system size. On petascale systems, which are I/O bound, this can cause a heavy overhead on the runtime of the application.

*Application-level checkpointing:* Application-level checkpointing [12], also called user-defined checkpointing, is a checkpointing method that enables the user to intelligently decide the placement and contents of the checkpointing. The primary advantage of this approach is that users can semantically understand the nuances of the applications and place checkpoints, using libraries and preprocessors, at critical areas, potentially decreasing the size of the checkpoint contents and checkpointing time. While this approach requires more programmer effort, it is more portable, since checkpoints can be saved in a machine-independent format, and thus offers better performance and flexibility as compared to the OS-level approach.

Compiler-level checkpointing [14] also exists. In this case, the compiler selects optimal checkpointing locations to minimize checkpointing content. Hybrid techniques, such as compiler assisted checkpointing coupled with application checkpointing, also have been studied. These provide a certain degree of transparency to the user.

Checkpointing in distributed systems requires that global consistency be maintained across the entire system and the domino effect be avoided. One way to achieve this consistency is through *coordinated checkpointing* [15]. In coordinated checkpointing, once the decision to checkpoint is made, the program does not progress unless all the checkpoints of all the processes are saved. Coordinated checkpointing requires that system or process components communicate with each other to establish checkpoint start and end times in single or multiple phases. Recovery in this technique is achieved by rolling back all processes to the latest state. Another method to achieve global consistency is through *independent checkpointing* [12], which uses synchronous and asynchronous logging of interprocess messages along with independent process checkpointing. In this method, recovery is achieved by rolling back to the faulty process and replaying the messages received by the faulty process. Other techniques such as *adaptive independent checkpointing* [12] are based on hybrid coordinated and independent checkpointing techniques.

Checkpointing optimizations have received considerable attention. Two primary techniques that have emerged are *full-memory checkpointing* and *incremental-memory checkpointing* [16]. In full-memory checkpointing, during each checkpoint instance the entire memory context for that process is saved. In incremental-memory checkpointing, pages that have been modified since the last checkpoint are marked as dirty pages and are saved. Incremental-memory checkpointing thus can reduce the amount of memory context that needs to be saved, especially for large systems.

## III. Overview of the Blue Gene/P System

The Blue Gene/P [7] [17] system at Argonne National Laboratory, named *Intrepid*, is IBM's massively parallel supercomputer consisting of 40 racks of 40,960 quad-core compute nodes (totaling 163,840 processors), 80 TB of memory and a system peak performance of 556 TF. Each compute node consists of four PowerPC 450 processors, operates at 0.85 GHz, and has 2 GB memory. Compute nodes run a lightweight Compute Node Kernel (CNK), which serves as the base operating system. In addition to the compute nodes, 640 input/output (I/O) nodes are used to communicate with the file system. The I/O nodes physically are the same as compute nodes but differ from them in functional. The Argonne system is configured to have a 1:64 ratio with a single I/O node managing 64 compute nodes. In addition to the compute and I/O nodes, there exist service and login nodes that allow diagnostics, compilation, scheduling, interactive activities, and administrative tasks to be carried out. This system architecture is depicted in Figure 1[1].

The Intrepid system supports five different networks. A three-dimensional torus network, based on adaptive cut-through routing, interconnects the compute nodes and carries the bulk of the communication while providing low-latency, high bandwidth point-to-point messaging. A collective network interconnects all the compute nodes and I/O nodes. This network is used for broadcasting data and forwarding file-system traffic to I/O nodes. An independent, tree-based, latency-optimized barrier network also exists for fast barrier collective operations. In addition, a dedicated Gigabit Ethernet and JTAG network that connects the I/O nodes and compute nodes to the service nodes is used for diagnostics, debugging

[1]This figure was taken from the 'IBM System Blue Gene/P Solution: Blue Gene/P Application Development' redbook [9]

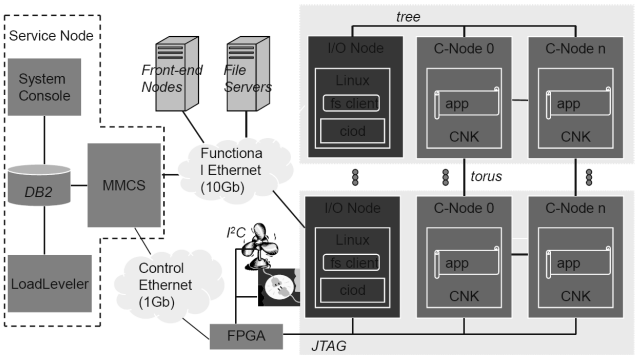


Fig. 1. The ANL BG/P System Architecture

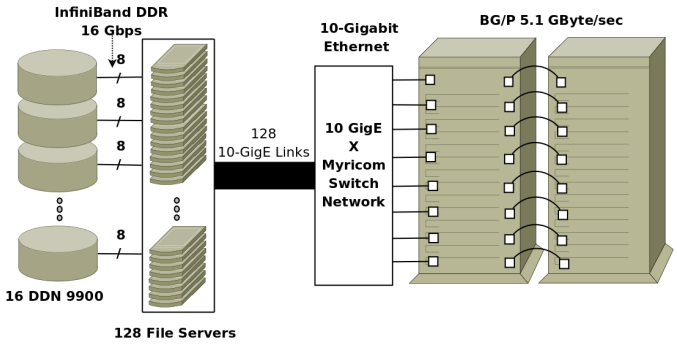


Fig. 2. The ANL BG/P File System

and monitoring. Lastly, a 10-Gigabit Ethernet network, consisting of Myricom switching gear connected in a nonblocking configuration, provides connectivity between the I/O nodes, file servers, and several other storage resources.

As seen in Figure 2, the Intrepid back-end file system architecture includes 16 DataDirect 9900 SAN storage arrays (each with 480 1 TB disks) offering around 8 petabytes of raw storage. Each array is directly connected to 8 file servers through 8 Infiniband DDR ports, each with a theoretical unidirectional bandwidth of 16 Gbps. The entire Intrepid system consists of 128 dual-core file servers, each having 8 GB memory each. These file servers, as earlier noted, connect to the 10-Gigabit Ethernet network through their 10-Gigabit Ethernet ports. The I/O nodes connect to this 10-Gigabit Ethernet network as well. We note that the peak unidirectional bandwidth of each 10-Gigabit Ethernet port of the I/O node is *limited to 6.8 Gb/s* by the internal collective network that feeds it [7]. From a theoretical performance standpoint, however, the network links connecting the file servers to the 10-Gigabit Ethernet network will become a bottleneck.

IBM has provided a special user-level checkpoint library [9] for BG/P applications. This library provides support for user-initiated checkpointing; a user can insert checkpoint calls manually at critical areas in the application. These calls are made available to the user through a checkpointing API provided by IBM. The IBM checkpointing library supports *full checkpointing*, where the *int BGCheckpoint()* library call can be used to take a snapshot of the program state at the instant at which it is called. All processes of the application should make this call to take a consistent global checkpoint. In addition to this call, the library provides several other calls to exclude regions from the program state, to call functions before or after the checkpoint takes place, and to restart applications from a certain checkpoint. Currently, the IBM checkpointing library provides no support for true incremental checkpointing.

## IV. Overview of the Applications Used

In this section, we describe the three applications from the fields of molecular dynamics and computational fluid dynamics, that we run on the IBM BG/P.

### A. Molecular Dynamics Simulations

In classical molecular dynamics [18], a single potential energy surface is represented by the force field. Usually. however, this level of representation is inefficient. More accurate representations, involving atomic structure, electron/proton distribution, chemical reactions, and electronic behavior can be generated by using quantum mechanical methods such as density functional theory (DFT). This area of molecular dynamics is known as ab initio (first principles) molecular dynamics (AIMD) [19]. GPAW (Grid Projected-Augmented method) and CPMD (Carr-Parrinello Molecular Dynamics) are two popular ab initio molecular dynamics codes. AIMD-based simulations tend to be highly complex, with more stringent requirements for computational and memory resources than traditional classical molecular dynamics codes have. The emergence of petascale machines such as the IBM Blue Gene series has resulted in significant provisions for running AIMD simulations efficiently and accurately. We thus choose two applications from this field, as described below, for our study of checkpointing trends on the IBM Blue Gene supercomputer.

*1) Grid-Based Projector-Augmented Wave Application:* The GPAW [20] application is a DFT-based code that is built on the projector-augmented wave (PAW) method and can use real-space uniform grids and multigrid methods. In the field of materials and quantum physics, the Schroedinger equation [21] is considered important because it describes how the quantum state of a physical system changes in time. Various electronic structure methods [22] can be used to solve the Schroedinger equation for the electrons in a molecule or a solid, to evaluate the resulting total energies, forces, response functions, atomic structure, electron distributions, and other attributes of interest. The PAW method is an electronic structure method for ab inito molecular dynamics with full-wave functions; and is often considered superior to other electronic structure methods. The GPAW application allows users to represent these pseudo wave functions on uniform real-space orthorhombic grids, enabling these modeling codes can be run on very large systems. The results of this can be used to provide a theoretical framework for interpreting experimental results and even to accurately predict the material properties before experimental data becomes available.

*2) Carr-Parrinello Molecular Dynamics Application:* The CPMD [23] [24] code is another electronic structure method and a parallelized plane wave/pseudo-potential implementation of density functional theory, which targets ab inito quantum mechanical molecular dynamics plane wave basis sets. The CPMD code is based on the Kohn-Sham [25] DFT code, and it provides a rich set of features that have been successfully applied to calculate static and dynamic properties of many complex molecular systems such as water, proteins, and DNA bases, as well as various processes such as photoreactions, catalysis, and diffusion. Reactions and interactions in such systems are too complicated to be handled by classic molecular dynamics; but they can be successfully handled in the Carr-Parrinello method because they are calculated directly from the electron structure in every time step. CPMD's flexibility and high performance on many computer platforms have made an optimal tool for the study of liquids, surfaces, crystals, and biomolecules

CPMD runs on many computer architectures. Its well-parallelized nature, based on MPI, makes it a popular application that can take advantage of petascale systems, motivating us to choose it for this study.

### *B. Nek5000*

Nek5000 [26] is an open source, spectral element computational fluid dynamics code developed at the Mathematics and Computer Science Division of Argonne National Laboratory. The C and Fortran code, which won a Gordon Bell prize, focuses on the simulation of unsteady incompressible fluid flow, convective heat with species transport, and magnetohydrodynamics. It can handle general two- and three-dimensional domains described by isoparametric quad or hex elements. In addition, it can be used to compute axisymmetric flows. Nek5000 is a time-stepping-based code and supports steady Stokes and steady heat conduction. It also features some of the first practical spectral element multigrid solvers, which are coupled to a highly scalable, parallel, coarse-grid solver.

The Nek5000 application was chosen for this study because it is highly scalable and can scale to processor counts of over 100,000, typical of petascale computing platforms. In addition, the Nek5000 software is used by many research institutions worldwide with a broad range of applications, including ocean current modeling, combustion, spatiotemporal chaos, the interaction of particles with wall-bounded turbulence, thermal hydraulics of reactor cores, and transition in vascular flows.

## V. Theory and Experiments

In this section we describe the experimental methodology used to conduct this study. We also describe our checkpointing model and use it in conjunction with the observed application memory trends to gain insight into optimal checkpointing parameters.

The scope of our study measures and analyzes the below trends.

1) How the application memory usage varies over application execution time and

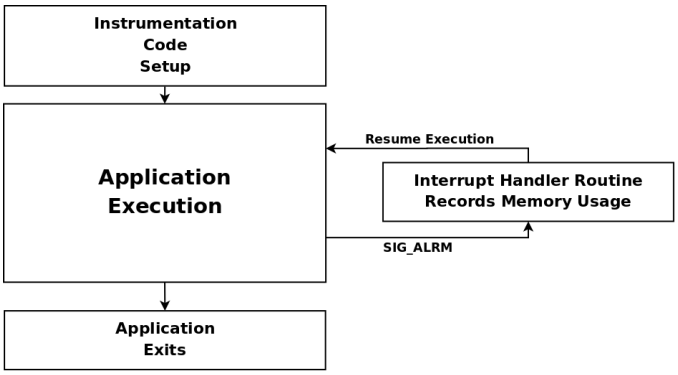


Fig. 3. Instrumentation Code

2) How the application memory usage varies with system size

These memory trends were observed for the GPAW, CPMD and Nek5000 applications for varying system size upto 32,768 cores. These trends were then analyzed to determine the checkpoint frequency and checkpoint duration using the optimum checkpoint model. In the next few subsections, we discuss the mechanism used to measure the memory usage and the details of our checkpointing model.

### *A. Recording Memory Usage Patterns*

In the 'Full memory checkpointing' technique, a snapshot of the entire process memory for every process is taken. In our experiments, we run the various application codes and record the data memory, stack memory and process-related information. Applications running on the IBM BG/P are statically linked, with the instruction-specific text code remaining remaining constant. We, thus, disregard the text sections of the program in our measurements. The data (which includes the heap memory) and stack memory, manage the local, global, static and declared variables, as well as memory requested by various system (new, malloc, calloc) calls - which constitutes the data that needs to be checkpointed.

In our setup, minimal amount of instrumentation code was inserted at the startup of each application that could record the memory usage for a process. An optimal method to achieve this, transparently to the application, would have been through *constructor* methods made available in a statically linked external library. However, the BG/P compilers, provided by IBM, do not provide this option. As an alternative, we invoke this measuring code soon after the entry point of the application.

From an implementation perspective, the memory-usage measuring code is based on a timer function that sets up a timer interrupt for a certain defined interval. As shown in Figure 3, the interrupt handler method which when invoked, measures the amount of memory used and records it. The *getrusage()* routine, available in the CNK operating system, is used to track the memory being used.

The getrusage function reports the resource usage statistics for the calling process or its child processes depending on

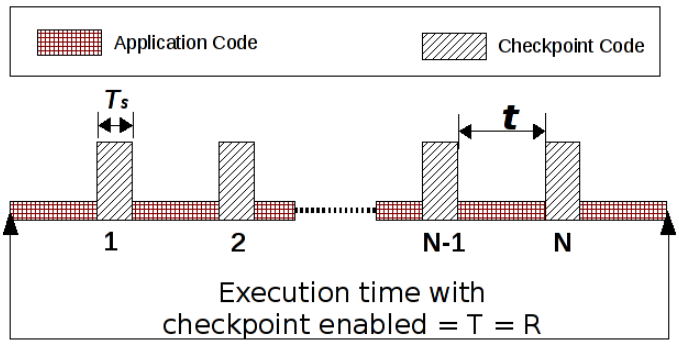


Fig. 4. Application Execution with Checkpoint Enabled

the parameters passed to the function. The following is the signature of the function: **int getrusage(int who, struct rusage *r_usage)**.

The *who* argument determines whether the values reported pertains the calling process or its child. Legitimate values for the *who* argument are RUSAGE_SELF which when used returns resource usage for the calling process and RUSAGE_CHILDREN which returns the waited-for or the terminated child process.

### *B. The Optimum Checkpointing Model*

Runtime on large systems like Intrepid is a valuable commodity, which users wish to use in the most optimal manner. While checkpointing is a necessary activity, users wish to devote only a certain percentage of application execution time to this procedure. Knowledge about checkpoint duration and frequency, based on such constraints, would help the user make more informed decisions and tradeoffs between their resource usage and application resiliency. We attempt to provide this information through the "optimum checkpoint model" discussed in this section. Our checkpoint model has been largely influenced by past research done with other checkpointing models [27] [28] in this field.

To arrive at the analytical model of the optimum checkpointing scheme on the Blue Gene/P system, let us consider an example application as shown in Figure 4. Like a majority of scientific applications, this application has a constant memory pattern for a majority of its execution period. The figure is a diagrammatic representation of the application with checkpointing feature enabled.

Here $T$ is the total execution time, including the time required to perform all the checkpoints; $T_s$ is the time required to complete one full checkpoint operation; $N$ is the optimum number of checkpoints to be performed during the length of the application execution; and $t$ is the optimum time interval between two checkpoints when the application resumes execution.

Let *n* be the number of cores the application is run on, *M* be the mean memory usage per core, and *B* be the unidirectional bandwidth from all the compute nodes to storage disks that is available to the entire application. Based on these parameters, the time required to save the state of all processes(cores) during a checkpoint can be given by:

$$T_s = \frac{M \times n}{B}. \tag{1}$$

If $T_A$ is the actual length of time the application needs to complete without checkpointing enabled, the total application runtime with checkpointing enabled i.e. $T$ is given by

$$T = T_A + (N \times T_s), \tag{2}$$

Given the total application execution time and the checkpoint time interval, the total number of optimum checkpoints can be derived by

$$N = \frac{T_A}{t} \tag{3}$$

Based on equation 2 and equation 3, we can derive the total application execution time:

$$T = T_A + \left(\frac{T_A}{t}\right) T_s$$

$$\Longrightarrow$$

$$T = T_A \left(1 + \frac{nM}{Bt}\right). \tag{4}$$

Let us assume that the user makes reservations for duration, $R$, close to the approximate total run time of the application, that is, $R \geq T$. If $X\%$ is the percentage of reserved time the user wishes to dedicate to perform checkpointing, then we have

$$X = \frac{R - T_A}{R}$$

$$T_A = (1 - X)R \tag{5}$$

Substituting from equations 1, 2 and 3 in equation 5 and assuming R = T , we have

$$R - N\left(\frac{M \times n}{B}\right) = (1 - X)R$$

Therefore *N* can be computed as follows:

$$\therefore N = \left\lfloor \frac{XRB}{nM} \right\rfloor \tag{6}$$

We can, thus, derive the number of optimum checkpoints based on the (a) percentage of reservation time dedicated for checkpointing, (b) bandwidth from the compute node to file servers and (c) the total amount of data to be checkpointed. Having this information provides users the flexibility to choose areas they wish to checkpoint. In practice, is may be difficult to predict the accurate value for *B* since several applications may use the I/O and network resources at the same time. However, the user can made educated guesses based on their system size and file system architecture, as we show in the next section. In addition, in reality, reservation times (*R*) are

rarely equal to the application execution time. However, for long running applications on large systems, end-users tend to have some information on the total execution times based on historical data or past runs performed, which can be used towards determining a more accurate reservation time.

Finally, substituting for $T_A$ in equation 3 and solving for $t$, we can get the time interval between two checkpoints, as follows:

$$t = \frac{M}{B}\left(\frac{n}{X} - 1\right) \tag{7}$$

Having checkpoint interval information makes it easier to develop independent libraries that can be compiled into the application and can used timer-based methods to checkpoint periodically.

### C. Application Evaluation

In this section, we carry our the evaluation of the GPAW, CPMD and the Nek5000 application on the Intrepid system.

Applications on the Intrepid system can be run in three modes: (a) SMP mode: In this mode, only one process, with a maximum of four threads can be launched on each compute node with each thread using a core, (b) Dual mode: In this mode, two processes with a maximum of two threads each can be launched with each thread using a core. The 2 GB is equally divided between the two processes; (c) Virtual mode: In this mode, a single process per core (i.e four processes per compute node) can be launched. The 2 GB memory is equally divided among all the four processes.

The GPAW application (version 0.4), being evaluated, consists of 256 water molecules with 2,048 electrons, 1,056 bands and $112^3$ grid points, with a grid spacing of 0.18. This application was run on the Intrepid system, with compute node count varying from 32 cores to 1,024 cores. The application was run in the smp mode with single thread. Figure 6(a) shows the memory trends for upto 256 cores and Figure 6 shows it for 512 and 1,024 cores. The x-axis shows the total application execution time and the y-axis shows the memory usage per core. As can be seen from the graphs, the memory requirements for the GPAW application grows relatively slowly against the total time execution as the system size increases. We also observe that the application execution time decreases with increasing system size. The memory footprint per core decreases with system size; however, the total application footprint (memory footprint per core * number of cores) increases with system size. The memory usage per core remains constant once the peak is attained, indicating a constant possible checkpointing time in later stages of the code.

Figure 6(a) shows memory trends for the CPMD application. The CPMD application was run in the SMP mode with four threads on a system size of 8,192 cores. While the CPMD memory consumption trend is similar to the GPAW application, we notice that CPMD memory consumption increases a little slowly as system size increases. The important difference is that the memory consumption for the majority of the execution time remains the same irrespective of the system size.

Figure 6(b) shows the memory consumption trend for the Nek5000. The Nek5000 application was run in virtual mode on a system size ranging from 8,192 cores to 32,768 cores. The 3-D graph in Figure 6(b) shows system size on the z-axis and the application execution time and memory usage (for the various system size), on the x-axis and y-axis respectively. In Nek5000, the memory is allocated as soon as the application starts up and remains the same for the execution lifetime of the process. Like CPMD, Nek5000 exhibits that the memory consumption does not vary with a change in system size.

### D. Computing Optimum Checkpointing Values

We briefly discussed the I/O and the network infrastructure of the Intrepid system in Section III. As discussed, the bandwidth between the 10-Gigabit Ethernet network and the file servers is theoretically 2 Tbps. When an application uses the entire BG/P system, all the 640 I/O nodes can theoretically deliver up to a maximum of 4.25 Tbps. This indicates that the bandwidth bottleneck for the maximum data throughput lies more towards the file servers than the I/O nodes when the system is running full capacity. However since our study involves only using upto 8 racks (32,768 cores)) out of the 40 racks (163,840 cores) available, we only use a maximum of 128 I/O nodes, thus limiting the I/O node bandwidth to 870 Gbps (i.e. 128 x 6.8Gbps)

Moreover, as we are utilizing only a fraction of the total system capacity, the available file server I/O bandwidth for our runs is also limited by the other applications that are performing I/O bound operations. Each I/O node is equipped with network interface capable of delivering 6.8 Gb/s. We based our optimum checkpoint value calculation on two cases. In the first case($B_{25}$), we assume that we are able to makes use of 25% of the total bandwidth provided by each I/O node interface. In the second case($B_{70}$), we optimistically assume that we make use of 70% of the total bandwidth provided by each I/O node interface.

On the Intrepid system, the I/O node to compute node ratio is 1:64. Each compute node consists of a quad-core processor. When an application is run in the virtual node mode or SMP mode with four threads, the 64 compute nodes provide a total of 256 cores. The Nek5000 was run in virtual node mode and CPMD application was run in SMP mode with 4 threads.

Assuming, $B_{I/O}$ to be the total bandwidth available per I/O node and *n* to be the number of cores the application is running on , the total bandwidth available to the Nek5000 application and the CPMD application can be computed by:

$$B = \left(\frac{n}{64 \times 4}\right) \times B_{I/O}. \tag{8}$$

The GPAW application is executed in SMP mode with one thread, with only core being used on each compute node. The total bandwidth available to the GPAW application can be computed by:

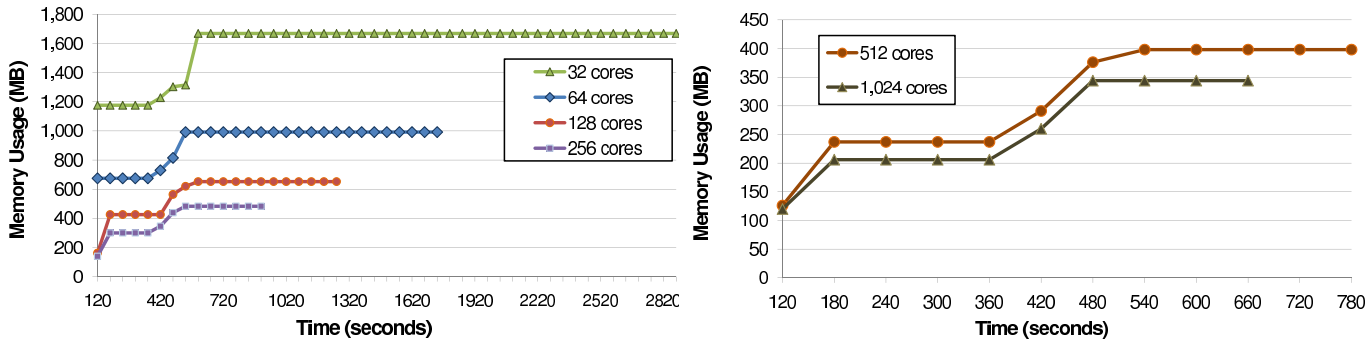


Fig. 5. GPAW Memory Consumption (a) Small Systems; (b) Large Systems

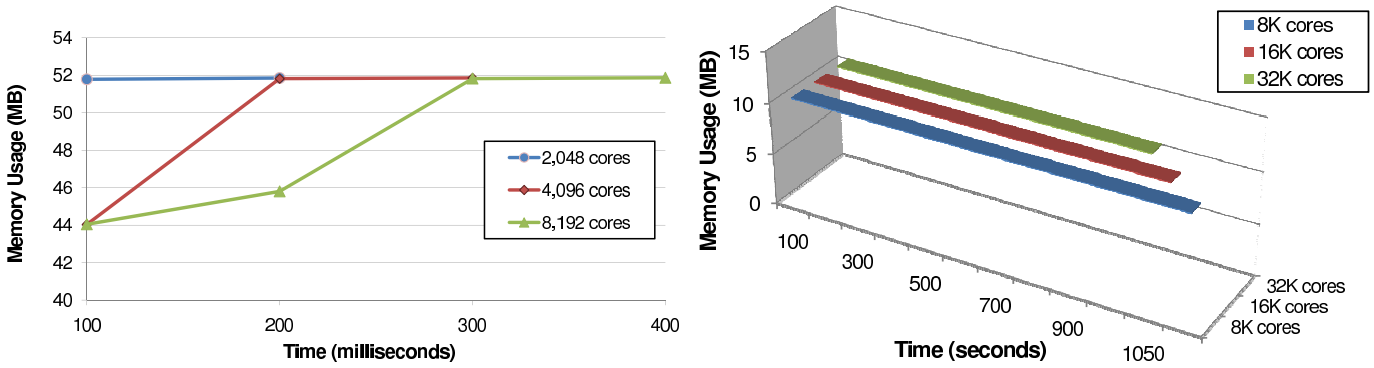


Fig. 6. (a) CPMD Memory Consumption (b) Nek5000 Memory Consumption

| n | M | R | 25% Bandwidth | | | 70% Bandwidth | | |
|---|---|---|---|---|---|---|---|---|
| | | | BW ($B_{25}$) | $N_{25}$ | $t_{25}$ | $B_{70}$ | $N_{70}$ | $t_{70}$ |
| 8192 | 10.52 | 1080 | 6800 | 21 | 50.69 | 19040 | 167 | 6.47 |
| 16384 | 10.52 | 1050 | 13600 | 20 | 50.69 | 38080 | 162 | 6.47 |
| 32768 | 10.52 | 1000 | 27200 | 19 | 50.69 | 76160 | 154 | 6.47 |

TABLE I
COMPUTED VALUES FOR NEK5000

$$B = \left(\frac{n}{64}\right) \times B_{I/O}. \quad (9)$$

For case 1 with 25% of total bandwidth available, we have $B_{I/O} = 0.25 \times 6.8Gb/s = \frac{0.25 \times 6800}{8} = 212.5$ MB/s. For case 2 with 70% of total bandwidth available, we have $B_{I/O} = 0.70 \times 6.8Gb/s = \frac{0.70 \times 6800}{8} = 595$ MB/s.

Based on the the equations from Section V-B and the observations in Section V-C we can compute the optimum checkpointing parameter values for each of these applications.

Let us assume that the user is willing to dedicate 5% of the reservation time for performing checkpoint operations. So we have $X = 0.05$.

Based on equations 6, 7, 8 and 9 and the graphs obtained in the previous section, we compute the optimum checkpointing parameters for the applications and tabulate them. In general we consider the reservation time $R$ to be about $10\%$ greater than the actual application run time obtained from the graphs.

Table I–III show the calculated checkpointing values for the three applications; where *n* is the number of cores, *M* is the average memory usage per core, *R* is the reservation time, *B* is the unidirectional bandwidth from all the compute nodes to storage disks, *N* is the number of optimal checkpoints, *t* is the checkpoint interval.

## VI. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed checkpointing trends for applications running on leadership class machines such as the IBM Blue Gene/P Intrepid system at Argonne National Laboratory. Ranked #5 on the Top500 November 2008 ranking, the Intrepid has 163,840 processors and a peak performance of 557 teraflops. We also presented an analytical model for efficiently computing the optimum checkpoint frequencies and intervals and studied three applications: the Grid-Based Projector-Augmented Wave application (GPAW), the Carr-Parrinello Molecular Dynamics application (CPMD), and a Nek5000 computational fluid dynamics application. We also showed

| n | M | R | 25% Bandwidth | | | 70% Bandwidth | | |
|---|---|---|---|---|---|---|---|---|
| | | | ($B_{25}$) | $N_{25}$ | $t_{25}$ | $B_{70}$ | $N_{70}$ | $t_{70}$ |
| 32 | 1500 | 3168 | 106.25 | 1 | 1792.94 | 297.5 | 13 | 225.45 |
| 64 | 1000 | 1914 | 212.5 | 1 | 1200 | 595 | 12 | 151.98 |
| 128 | 650 | 1386 | 425 | 1 | 781.53 | 1190 | 13 | 99.33 |
| 256 | 475 | 990 | 850 | 1 | 571.68 | 2380 | 13 | 72.79 |
| 512 | 400 | 858 | 1700 | 1 | 481.65 | 4760 | 13 | 61.38 |
| 1024 | 350 | 726 | 3400 | 1 | 421.54 | 9520 | 13 | 53.74 |

TABLE II
COMPUTED VALUES FOR GPAW

| n | M | R | 25% Bandwidth | | | 70% Bandwidth | | |
|---|---|---|---|---|---|---|---|---|
| | | | BW ($B_{25}$) | $N_{25}$ | $t_{25}$ | $B_{70}$ | $N_{70}$ | $t_{70}$ |
| 2048 | 51.82 | 220 | 1700 | 0 | 249.68 | 4760 | 6 | 31.84 |
| 4096 | 51.85 | 330 | 3400 | 1 | 249.84 | 9520 | 10 | 31.86 |
| 8192 | 51.87 | 440 | 6800 | 1 | 249.94 | 19040 | 13 | 31.88 |

TABLE III
COMPUTED VALUES FOR CPMD

with the help of experimental data and computed values how application scaling characteristics influence checkpoint-related decisions. Our current work considered "full checkpointing",where the entire program state of the processes is stored during the checkpoint operation. We chose this approach because the IBM checkpointing library currently supports only full checkpointing. We are conducting a similar study for incremental checkpointing of applications on IBM BG/P, which will be useful for incremental checkpointing libraries built in the future for this machine. Our current study has been conducted on upto 32,768 processes of the Intrepid system.

## REFERENCES


[1] J. Dongarra, H. W. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 11th Edition. Technical Report UT-CS-98-391, 1998. Available at http://citeseer.ist.psu.edu/dongarra94top.html, Also refer http://www.top500.org/.

[2] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks RAID. In *ACM SIGMOD International Conference on Management of Data*, 1988.

[3] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In *ACM Fall joint computer conference*, 1986.

[4] J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. In *Software: Practice and Experience*, volume 29-2, 1999.

[5] G. Bronevetsky, D. Marques, K. Pingali, and R. Rugina. Compiler-enhanced incremental checkpointing. In *LNCS series: Languages and Compilers for Parallel Computing*, 2008.

[6] S. Garg, Y. Huang, C. Kintala, and K. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1996.

[7] IBM Blue Gene Team. Overview of the IBM Blue/Gene Project. 52-1/2, 2008.

[8] J. Sancho, F. Petrini, G. Johnson, and E. Frachtenberg. On the feasibility of incremental checkpointing for scientific computing. In *IPDPS*, 2004.

[9] C. Sosa and B. Knudson. IBM System Blue Gene/P Solution: Blue Gene/P Application Development. 2007. Available at http://www.redbooks.ibm.com/abstracts/sg247287.html.

[10] J. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, University of Tennesse, Knoxville, 1997. Available at http://www.cs.utk.edu/ plank/plank/papers/CS-97-372.html.

[11] L. Silva and J. Silva. System-level versus user-defined checkpointing. In *SRDS*, 1998.

[12] A. Zomaya. Parallel and distributed computing handbook. 1995.

[13] G .Barigazzi and L. Strigini. Application-transparent setting of recovery points. In *FTCS*, 1983.

[14] C. Li and W. Fuchs. Catch - compiler assisted techniques for checkpointing. In *FTCS*, 1990.

[15] C. Guohong and M. Singhal. On Coordinated Checkpointing in Distributed systems. In *TPDS*, volume 9-12, pages 1213–1225, 1998.

[16] J. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, 1995.

[17] N. Desai, R. Bradshaw, C. Lueninghoener, A. Cherry, S. Coghlan, and W. Scullin. Petascale system management experiences. In *LISA*, 2008.

[18] R. Car, M. Parrinello, J. Schmidt, and D. Sebastiai et al. Unified approach for molecular dynamics and density-functional theory. 1985.

[19] D. Marx and Hutter J. Ab-initio Molecular Dynamics: Theory and Implementation. NIC series, Volume 3, 2000.

[20] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71(3):035109, Jan 2005.

[21] T. Cazenave. An Introduction to Nonlinear Schrodinger Equations. 1989.

[22] J. Foresman and E. Frisch. Exploring chemistry with electronic methods. 1996.

[23] D. MArx and J. Hutter. Ab-initio molecular dynamics: Theory and implementation. *Modern Methods and Algorithms of Quantum Chemistry Forschungzentrum Juelich, NIC Series*, 1, 2000.

[24] Wanda Andreoni and Alessandro Curioni. New advances in chemistry and materials science with cpmd and parallel computing. *Parallel Comput.*, 26(7-8):819–842, 2000.

[25] W. Koch and M. Holthausen. A chemist's guide to density functional theory. 2001.

[26] James W. Lottes and Paul F. Fischer. Hybrid multigrid/schwarz algorithms for the spectral element method. *Journal of Scientific Computing*, pages 45–78, 2004.

[27] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.

[28] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.